DYNAMO: AMAZON’S HIGHLY AVAILABLE KEY-VALUE STORE

Presented by Byungjin Jun
What is Dynamo for?

- Highly available key-value storages system
  - Simple primary-key only interface
  - Scalable and Reliable
    - Tradeoff: Consistency
  - Guarantee Service Level Agreements (SLA)
System Assumptions and Requirements

• Query Model
  • Read / Write operations
  • Uniquely identified by a key

• ACID Properties
  • Atomicity, Consistency, Isolation, Durability

• Efficiency
  • Latency is measured at the 99.9th percentile of distribution

• No security issue
SLA

- A client & a service
  - Client’s expected request rate distribution for a particular API
  - Expected service latency

- Amazon
  - All customer’s satisfaction
  - 99.9th percentile of the distribution
  - Performance vs. Cost-effectiveness
Design Consideration

• Availability vs. Strong consistency
  • Replicas

• Resolving update conflicts
  • When: during read. “Always writable”
  • Who: by the application. “Merge”

• Other key principles
  • Incremental scalability
  • Symmetry
  • Decentralization
  • Heterogeneity
### Techniques in Dynamos & Advantages

<table>
<thead>
<tr>
<th>Problem</th>
<th>Technique</th>
<th>Advantage</th>
</tr>
</thead>
<tbody>
<tr>
<td>Partitioning</td>
<td>Consistent Hashing</td>
<td>Incremental Scalability</td>
</tr>
<tr>
<td>High Availability for writes</td>
<td>Vector clocks with reconciliation during reads</td>
<td>Version size is decoupled from update rates.</td>
</tr>
<tr>
<td>Handling temporary failures</td>
<td>Sloppy Quorum and hinted handoff</td>
<td>Provides high availability and durability guarantee when some of the replicas are not available.</td>
</tr>
</tbody>
</table>
## Techniques in Dynamos & Advantages (cont.)

| Problem                        | Technique                                      | Advantage                                                                 |
| narrowed_list | narrowed_list | narrowed_list | narrowed_list |
|-------------------------------|------------------------------------------------|---------------------------------------------------------------------------|
| Recovering from permanent failures | Anti-entropy using Merkle trees | Synchronizes divergent replicas in the background. |
| Membership and failure detection | Gossip-based membership protocol and failure detection. | Preserves symmetry and avoids having a centralized registry for storing membership and node liveness information. |
Partitioning Algorithm

- Scale incrementally
  - Dynamic partitioning

- Consistent hashing
  - Ring-based
  - Locality: node’s position & hashed key
  - Problem?

- Virtual nodes
  - Each node can be responsible for more than one virtual node
  - Make load dispersed equivalently
  - The number of virtual node & capacity of a node
Replication

• Replication of data
  • N nodes

• Coordinator node
  • In charge of the replication of the data items
  • The first among the top N nodes

• Preference list
  • List of nodes storing a particular key
  • More than N nodes
  • Skip some positions to contain distinct physical nodes.
Data Versioning

• Problem in update
  • Put() can return before update
  • Get() can return object with not latest update

• Version
  • A result of modification -> new and immutable version
  • Reconciliation
    • Syntactic reconciliation
      • New version subsume the previous version
    • Semantic reconciliation
      • Version branching -> conflicting versions of the same object
      • Collapse multiple branches of data evolution back into one
      • Merging
Data Versioning (cont.)

- Vector clock
  - Capture causality
  - A list of (node, counter) pair
  - Every version of every object
  - Comparison of counters

- A possible issue
  - The size of vector clocks if many servers writes to an object
  - Clock truncation scheme
    - Timestamp
    - Over threshold, remove older one
Execution of get() and put() operations

- Selecting a proper node
  - Load balancer
    - based on load information
    - Client doesn’t have additional information
    - Forward requests to any random node
  - Partition-aware client library
    - Choose the appropriate coordinator nodes
    - Lower latency: no forwarding steps

- Operation of Read / Write
  - First N healthy nodes in the preference list
  - Node failure / network partition -> find lower ranked node
Execution of get() and put() operations (cont.)

- **Sloppy Quorum**
  - Consistency protocol
  - R / W: a minimum number of nodes that must participate in a successful read/write operation
    - W = 1: never reject a write if there is at least one node can process it
    - R = 1, W = N: high performance read engine. Rare update.
    - Low R / W possibly cause inconsistency

- R + W > N -> quorum-like system
- The slowest R / W replica decides latency
- N > R, W
  - Better latency
- Common (N, R, W) = (3, 2, 2)
Handling failures

• Hinted Handoff

• Assume N=3, B is temporarily down during a write
  • Send hinted replica to E
  • Metadata: have a hint
    • Belong to B
    • If B is recovered, send it back

• Replicated across multiple data centers
Handling permanent failures

- Replica synchronization (anti-entropy)

- Merkle hash tree
  - Leaves: hashes of the values of individual keys
  - Parent node: hash of their respective children
    + Each branch is checked independently w/o entire tree
    + Reduce the amount of data to be transferred (root comparison)
      – Many key ranges change when a node join/leave

- Each node maintains Merkle tree
Membership and Failure Detection

- **Ring membership**
  - Gossip-based protocol
  - Token set and corresponding nodes

- **External Discovery**
  - Prevent logical partition
    - Membership of other nodes
  - Seeds: know all nodes, prevent logical partitions

- **Failure Detection**
  - No response = failed
  - Periodic check for the recovery
Balancing performance and Durability

• Performance
  • R / W
  • $99.9^{th}$
  • 200mse
Balancing performance and Durability (cont.)

• Buffering
Ensuring Uniform Load distribution

• Load imbalance
  • Load & imbalance ratio
Ensuring Uniform Load distribution (cont.)

- Dynamo’s Partitioning schemes
  - Strategy 1: \( T \) random tokens / node and partition by token value
  - Strategy 2: \( T \) random tokens / node and equal sized partitions
    - Size: \( Q \) (\( Q \gg N, Q \gg S \times T \). \( S \): the number of nodes in the system)
  - Strategy 3: \( Q/S \) tokens / node, equal sized partitions
Ensuring Uniform Load distribution (cont.)

- Comparison of the load distribution efficiency
  - $S = 30$
  - $N = 3$
  - $3 > 1 > 2$

- Strategy 3
  + Faster recovery/bootstrapping
  + Ease of archival
  - Coordination required
Client-driven vs. Server-driven Coordination

- Only N nodes in preference list can write.

- Alternative approach
  - State machine -> client nodes
  - Client downloads membership state from a random node
  - Load balancer is no longer required
  - Eliminates the overhead of the load balancer and extra network hop

<table>
<thead>
<tr>
<th>latency(ms)</th>
<th>99.9th read</th>
<th>99.9th write</th>
<th>Average read</th>
<th>Average write</th>
</tr>
</thead>
<tbody>
<tr>
<td>Server</td>
<td>68.9</td>
<td>68.5</td>
<td>3.9</td>
<td>4.02</td>
</tr>
<tr>
<td>Client</td>
<td>30.4</td>
<td>30.4</td>
<td>1.55</td>
<td>1.9</td>
</tr>
</tbody>
</table>
Other features

• Divergent Versions
  • Failures: node failures, data center failures, network partitions
  • Concurrent large number of writes and multiple node update
  • Very rare: 99.94% of requests saw one version.

• Balancing background vs. foreground tasks
  • Background tasks affect performance
  • Admission control mechanism
    • Monitor resources while executing a foreground put/get operations
    • Assign resources to background tasks
Conclusion

• Dynamo: highly available and scalable data store
• Performance & failures handling
• Service owner customize storage system: N, R, W
• Single highly-available system
  -> block of highly-available system